LMS Popular Lectures

Codes

Peter J. Cameron

p.j.cameron@qmul.ac.uk

June/July 2001

Think of a number . . .

Think of a number between 0 and 15.

Now answer the following questions.

You are allowed to lie **once**.

Think of a number . . .

Think of a number between 0 and 15.

Now answer the following questions.

You are allowed to lie **once**.

1. Is the number 8 or greater?
2. Is it in the set $\{4, 5, 6, 7, 12, 13, 14, 15\}$?
3. Is it in the set $\{2, 3, 6, 7, 10, 11, 14, 15\}$?
4. Is it odd?
5. Is it in the set $\{1, 2, 4, 7, 9, 10, 12, 15\}$?
6. Is it in the set $\{1, 2, 5, 6, 8, 11, 12, 15\}$?
7. Is it in the set $\{1, 3, 4, 6, 8, 10, 13, 15\}$?

Base 2 and modulo 2

Our familiar way of writing numbers uses *base ten*: for example

$$1354 = 1 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 4 \times 10^0.$$

We could use any number as a base. One which is important is *base 2*:

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0,$$

which is the number we usually write as 13 (using base ten).

Note that 10 (in base 2) is $2^1 = 2$; so we have $1 + 1 = 10$.

Later we will use another kind of addition, *modulo two*, where instead we have the equation $1 + 1 = 0$. This looks confusing but it is really easier than ordinary addition: there is no carrying!

How does the trick work?

If no lies were allowed, we would only need four questions. Any number in the range from 0 to 15 can be written in base 2 with four digits (starting with zero if necessary); for example,

$$5 \text{ (base 10)} = 0101 \text{ (base 2)}.$$

If we record the answers to the questions as either 0 (for ‘no’) or 1 (for ‘yes’), then the first four of our seven questions generate the base 2 representation. For they are equivalent to asking:

Is the 8s digit 1?
Is the 4s digit 1?
Is the 2s digit 1?
Is the units digit 1?

The last three questions are put in to catch the liars!

How does it work?

The possible answers to the seven questions, if no lies are told, are:

0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	1
2	0	0	1	0	1	1	0
3	0	0	1	1	0	0	1
4	0	1	0	0	1	0	1
5	0	1	0	1	0	1	0
6	0	1	1	0	0	1	1
7	0	1	1	1	1	0	0
8	1	0	0	0	0	1	1
9	1	0	0	1	1	0	0
10	1	0	1	0	1	0	1
11	1	0	1	1	0	1	0
12	1	1	0	0	1	1	0
13	1	1	0	1	0	0	1
14	1	1	1	0	0	0	0
15	1	1	1	1	1	1	1

Let C be the list of sixteen 7-tuples of zeros and ones in the table. C is our first example of a *code*. We use the notation v_n for the 7-tuple corresponding to the number n.

A vector space

We regard 0 and 1 as being the elements of the *binary field*.

That is, we add and multiply them *modulo two*, that is, according to the following rules:

$+$	0	1
0	0	1
1	1	0

$\times$	0	1
0	0	0
1	0	1

We denote this field by $\mathbb{F}_2$.

Now the set of all 7-tuples of zeros and ones is a 7-dimensional *vector space* over the field $\mathbb{F}_2$. Two vectors are added component by component: for example,

$$0010100 + 1110001 = 1100101.$$

Now some diligent checking shows that C *is closed under addition*. For example, $v_{13} + v_{11} = v_6$.

Distance

Further checking shows that any two elements of C have different entries in at least three positions. Since the distance between v and w is just the number of ones in $v+w$, we just have to observe that all non-zero codewords have at least three ones.

For example,

$$v_4 = 0100101$$

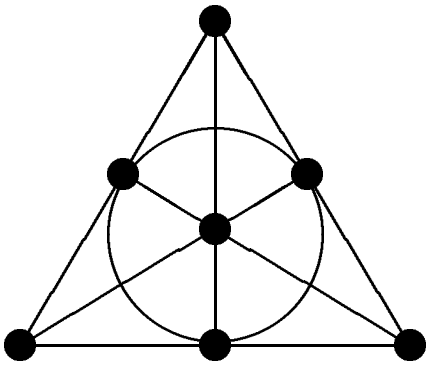
and

$$v_9 = 1001100$$

differ in the first, second, fourth and seventh positions, since $v_4 + v_9 = v_{13} = 1101001$.

If you tell a lie, then your answers to the questions will only differ in one place from the correct answers. This means that they will differ in at least two places from the correct answers for any other number. So *in principle* I can work out which number you were thinking of and which answer is a lie.

A projective plane



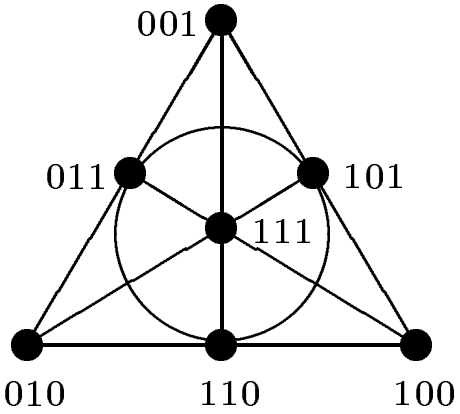
This picture shows a projective plane.

It has seven points and seven lines.

Any line contains three points, and any point lies on three lines.

Any two points lie together on just one line, and any two lines meet in just one point (that is, there are no parallel lines).

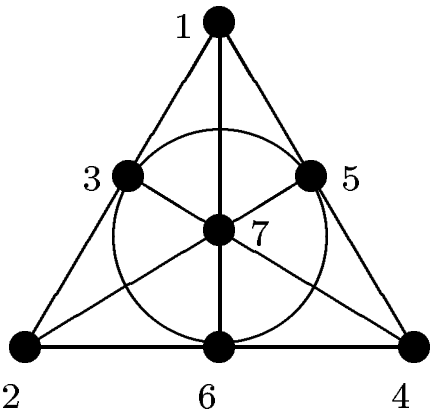
The plane with coordinates



We can give coordinates to the points of the plane. Notice that the coordinates of the points on any line add up to 000 (where the addition is binary, as usual).

We can also regard these coordinates as being the numbers from 1 to 7, written in base 2. Then we get the picture on the next slide.

The plane with numbers



Now something remarkable happens.

The code we had earlier has 16 words. One is all-zeros, and one is all-ones. Of the others, seven have three zeros and four ones, and the lines of the plane give the positions of the three zeros. The other seven have three ones and four zeros, and the lines of the plane give the positions of the three ones.

If you have played the game of Nim, you will recognise these as winning positions.

The game of Nim

The game of Nim is played with matches. Put any number of piles of matches on the table, with any number of matches in each pile. Two players take turns to remove matches from the table; each player can remove any positive number of matches from *one pile*. The player who takes the last match wins.

To work out a winning strategy, you do the following:

- Convert the numbers of matches in the piles to *base* 2.
- Add these numbers *modulo* 2.
- If the answer is zero, the player who just played can win; otherwise, the other player can win.

For example, $\{3,5,6\}$ is a winning position for the player who just played, since
$(011)+(101)+(110)=(000)$.

How the trick really works

With this information, you can do the decoding in your head.

Look at the responses to the questions. If they are all 0s, then no lies were told. If there was just one 1, then it was the lie.

If there were two 1s in positions i and j, find the third point k on the line through i and j; this is the position of the lie.

If there are three 1s which form a line of the plane, no lie was told. If there are three 1s which do not form a line, then the positions of the zeros contain just one line; the odd point out is the lie.

If there are more 1s than 0s, just reverse zeros and ones.

For example, if the response is 0111000, then 234 is not a line, but 1567 contains the line 167; so 5 is the lie, the correct response is 0111100, and the number is $0111 = 7$.

Codes

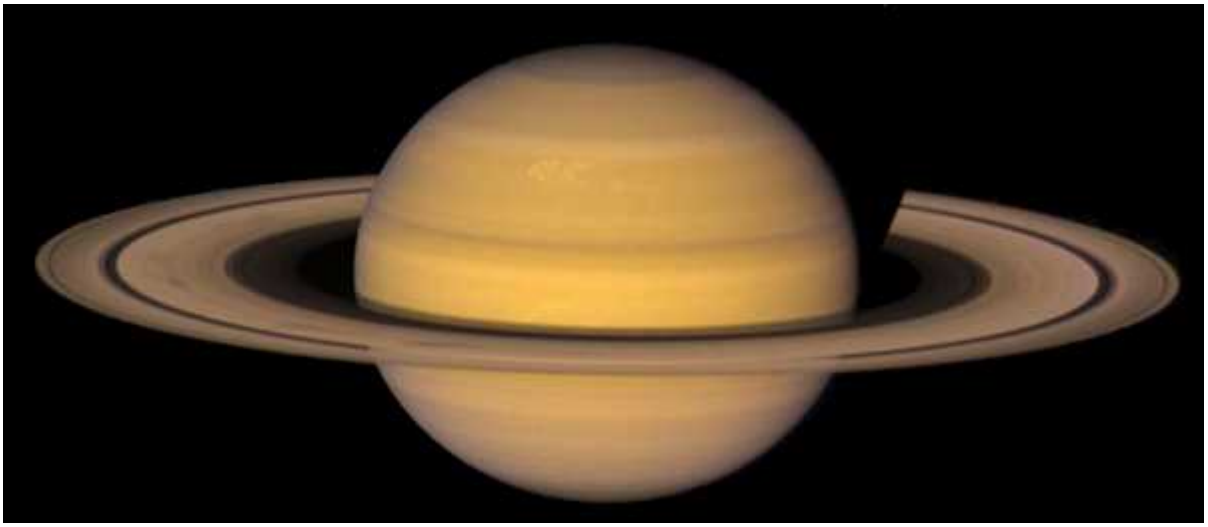
Now we turn this into mathematics. Consider an alphabet A of symbols. (In our example, A is the *binary alphabet* $\{0,1\}$). A *code* is simply a set of words of length n, or n-tuples, of elements of the alphabet A.

The *minimum distance* of the code is the smallest distance between two different codewords. If the minimum distance is at least $2e+1$, then the code can correct up to e errors. (We had $e=1$ in our example.)

Encoding and decoding are easier if the alphabet A is a field, and the sum of two codewords (or product of a codeword by a scalar) are again in the code. In this case, we say that the code is *linear*.

Codes are used in many practical situations. We now look at a few of these.

The planets



Error-correction is used for getting pictures and data about the Solar System back to earth.

For example, the Voyager spaceprobes had a 400-watt power supply to drive all the instruments; the transmitter used only 30 watts! Information was transmitted over hundreds of millions of kilometers of space. Errors occur during transmission: this is ‘nature lying to us’, and we can use a code to get round this. The above picture used a famous code called the *Golay code*, which also features in some of the latest developments in group theory and mathematical physics.

Compact discs

A compact disc holds music (or data, in the case of a CD-ROM) in digital form. A typical CD comes off the production line with about $500\,000$ bit errors, and after normal use may have $1\,000\,000$ bit errors. Many of them are *burst errors*, i.e. a scratch destroys a long run of bits. How can we correct burst errors efficiently?

Take a large alphabet A made up of, say, 2^{16} symbols. Each symbol can be encoded as a string of 16 bits.

Now suppose we have a code which corrects one error over the alphabet A. Up to 16 bit errors in the output string will only cause a single symbol of A to be received incorrectly, and so can be corrected.

Thus the code cannot correct more than one error ‘scattered about’, but can correct up to 16 errors occurring ‘in a burst’.

Finite fields

We'd like the alphabet to be a field. A theorem of Évariste Galois (the French mathematician who died in a duel in 1832 at the age of 21) says that we can find a field with 2^{16} elements.

The theorem of Galois says that there is a finite field with any given *prime power* number of elements.

Electrical engineering books often contain tables of large finite fields.

The Human Genome Project

The human genome has now been sequenced. How did they do it?

It is not possible to run a molecule of DNA through a machine and read off the sequence of bases. Instead, the molecule is chopped into a lot of small pieces. Now short sequences of bases can be recognised, so we can test the pieces to see if such sequences occur. Two sequences which occur together on the same piece must be close together on the original molecule. This information is then patched together to get the sequence.

If the pieces are short, we can assume that at most one contains any two given sequences, and we want to identify which piece does so.

Let us make a mathematical model which captures the essence of this.

A combinatorial search problem

We are given a set of n objects, containing one 'active pair'.

We can test any subset: the test is positive precisely when the subset contains both members of the active pair.

How many tests are required to identify the active pair? (We want to use as few tests as possible.)

We could test all $n(n-1)/2$ pairs. More economically, we can find a set of about $2n$ subsets such that each pair is contained in at least two of them, and any two have at most one pair in common; then testing these subsets will find the pair.

Using coding theory

We can do much better with a 2-error-correcting code.

Imagine that a message is transmitted consisting entirely of zeros, but an error occurs in the transmission and two zeros are changed into ones (in the positions of the active pair). A 2-error-correcting code can find the positions of the ones, by asking a number of questions.

Now each question is of the form we saw before, corresponding to a subset of the positions. Corresponding to each question, we perform two tests, one on the subset specified by that question, and one on the complementary subset. From the results we can get the information needed by the code to locate the errors.

In this way, we can locate the active pair in only about $4 \log_2 n$ tests.

Bonuses

Of course, the method easily adapts to the case where we are looking for a triple, quadruple, etc., of elements, rather than a pair.

There is an extra bonus too.

Suppose that a small proportion of test results may be incorrect. We can think of this as ‘nature lying to us’ and use another code to catch the lie.

For example, suppose that $n = 1000$. There is a 2-error-correcting code which will solve the problem in 40 tests. If some of the test results are wrong, but not more than 3% in a given run, then we can still identify the active pair in 60 tests.

The codes in this example are called *BCH codes*, after R. C. Bose, D. K. Ray-Chaudhuri, and A. Hocquenghem.

Quantum computing

Quantum computing may be one of the exciting developments of the new century. Nobody has managed to build a quantum computer yet, but in theory it could perform certain tasks much faster than a conventional computer. An example of such a task is factorising a large number into its prime factors. (This is thought to be a difficult problem, and the security of Internet communication depends on the assumption that nobody can factorise large numbers quickly).

A quantum computer could potentially do this because it can carry out many different calculations at the same time, because of the *superposition principle*.

In a quantum computer, a bit of information is stored by a single nuclear spin. So the machine is very vulnerable to errors caused by interference from the environment.

Quantum error correction

Quantum theory is based on *complex numbers*, which have both modulus and argument. So they are vulnerable to both *bit errors* (e.g. changing a 0 to a 1) and *phase errors* (changing the argument of the complex number).

To correct these, we use codes based on the Galois field $\mathbb{F}_4$ with four elements. Here, ω is a cube root of unity, and $\overline{\omega}$ its 'complex conjugate'. We encode two bits (a,b) as the element $a\omega + b\overline{\omega}$.

$+$	0	1	ω	$\overline{\omega}$
0	0	1	ω	$\overline{\omega}$
1	1	0	$\overline{\omega}$	ω
ω	ω	$\overline{\omega}$	0	1
$\overline{\omega}$	$\overline{\omega}$	ω	1	0

$\cdot$	0	1	ω	$\overline{\omega}$
0	0	0	0	0
1	0	1	ω	$\overline{\omega}$
ω	0	ω	$\overline{\omega}$	1
$\overline{\omega}$	0	$\overline{\omega}$	1	ω

Picture credits

The logos and banners are the property of the organisations they represent.

The picture of Saturn is from the Planetary Data System at NASA JPL in Pasadena, California:

`http://pds.jpl.nasa.gov/planets/`

The picture of Galois is taken from the MacTutor History of Mathematics Archive at the University of St. Andrews:

`http://www-groups.dcs.st-and.ac.uk/~history/`

The lie detector picture is © Neill Cameron 2001:

`http://www.geocities.com/planetdumbass/default.html`